PGI® Parallel Compilers and Tools for x86 and AMD64 Workstations, Servers and Clusters

November 2003

Douglas Miles
STMicroelectronics
The Portland Group Compiler Technology
www.pgroup.com

Who the heck is STMicroelectronics and why did they buy The Portland Group?

- STMicroelectronics is the #3 semiconductor manufacturer as measured by worldwide revenues in 2002 – microcontrollers (printers, disk drives, automotive, radio), Flash memory, RF parts for cell phones, etc, etc

- ST made a commitment to develop a world-class compiler and tools infrastructure by buying The Portland Group (a.k.a. PGI) in December, 2000 and augmenting it with various existing internal resources and technologies

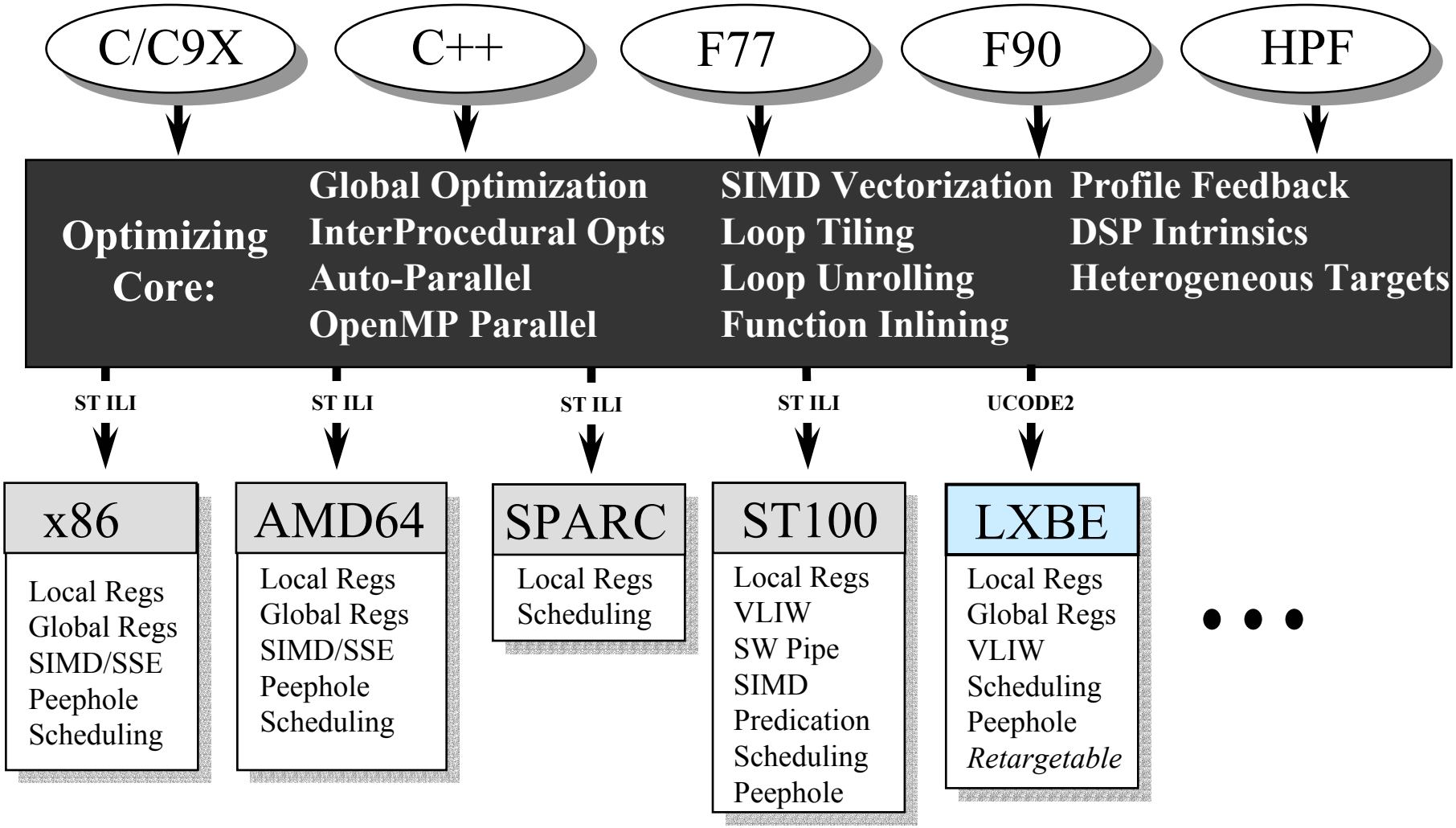
- ***Obvious Question:*** Why continue to develop AMD64 and x86 compilers and tools?
  Answer #1: How else do you measure "world-class"?
  Answer #2: Volume
  Answer #3: Much compiler technology first developed for HPTC has migrated down to embedded platforms; ST will specifically leverage this on an ongoing basis

- ***End result:*** it is part of our charter to develop what are demonstrably the highest-performance, highest-quality x86 and AMD64 compilers and tools, period. In this sense, our interests are perfectly aligned with Linux/x86/AMD64 end-users, and also well-aligned with AMD

Overview

- ST/PGI Compiler technology => PGI compiler products commercial (just a short one)
- Important compiler options
- Getting the right answers; x87 extended precision FP ops versus 32-bit and 64-bit IEEE SSE ops
- Streaming SIMD enhancements – SSE present and future
- Parallelization, what can you expect? OpenMP directive-based parallelization
- Benchmarks, Future enhancements, Q&A

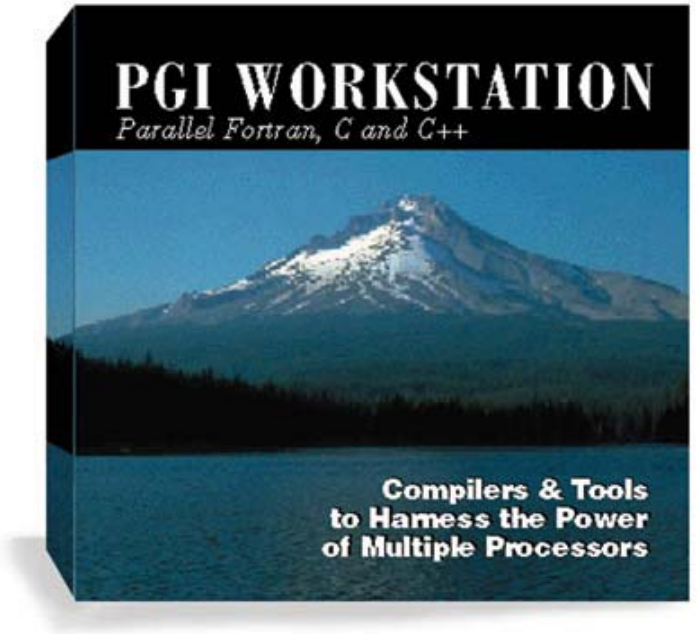
ST/PGI Optimizing Compilers



- Languages and optimizations enabled/disabled per target requirements
- LXBE is a Highly Optimizing easily Retargetable Code Generator

PGI® Workstation – 1 to 4 CPU Systems

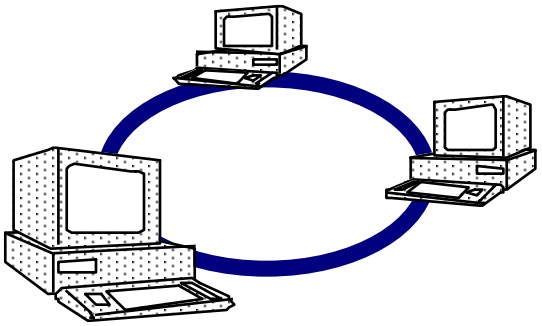
Compiler	Language	Command
PGF77®	FORTRAN 77	`pgf77`
PGF90™	Fortran 90	`pgf90`
PGHPF®	High Performance Fortran	`pghpf`
PGCC®	ANSI and K&R *C*	`pgcc`
PGC++™	ANSI *C++* with *cfront* compatibility features	`pgCC`
PGDBG®	Source code debugger	`pgdbg`
PGPROF®	Source code performance profiler	`pgprof`



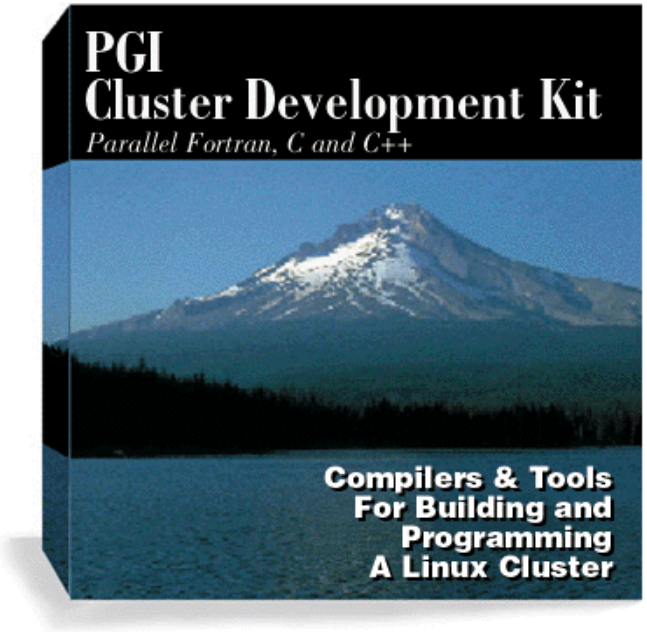
Linux *32-bit/64-bit* *Windows*

Pentium 4 *AMD Athlon™* *Xeon* *AMD Opteron ™*

PGI CDK ™ = PGI Compilers + Open Source Clustering Software



Workstation Clusters



The PGI CDK is a turn-key package for configuration of an HPC cluster from a group of networked Linux workstations or dedicated cluster using a simple question-and-answer installer

Why PGI?

- Highly optimized for *all* 32-bit x86 (Pentium4/Xeon/AMD Athlon™XP) and 64-bit AMD64 (AMD Opteron™/Athlon64) CPUs – we optimize our compilers and tools for end-users, not to sell processors
- The most comprehensive Linux support in the industry – Red Hat 6.0 – 9.0, SuSE 7.0 – 9.0, lower volume distributions; new distributions usually supported within a few months of release
- UNIX-heritage compilers => easy, incremental RISC/UNIX to Linux migration; easy, incremental 32-bit to 64-bit migration
- Support for 32-bit *and* 64-bit Linux86, Windows NT/2K/XP – PGI provides a uniform user interface across Linux and Win32
- Efficient native OpenMP support for all languages and tools
- *Infrastructure* – NAG, VNI, MPI-Pro, TotalView, VAMPIR, ISVs

Key Features

- Outstanding single-processor Fortran performance on *all* 32-bit x86 and 64-bit AMD64 processors
- Automatic use of vector SSE/SSE2 streaming SIMD instructions
- Inter-procedural analysis and optimization, function inlining
- Auto-parallel/OpenMP for multi-CPUs in F77, F90, C, C++
- EDG 3.1 ISO/ANSI C++ front-end
- RISC/UNIX compatibility – I/O, strings, symbols, options, etc
- Byte-swapping I/O for interoperability with RISC/UNIX
- GNU interoperability – *gcc* / *g77* / *gdb*
- Large file support (> 2GB) on 32-bit x86 systems
- ACML high-performance BLAS, LAPACK and FFT library
- *Infrastructure* – NAG, VNI, MPI-Pro, TotalView, VAMPIR, ISVs

Important PGI Compiler Options

`-O2/-O3`	Scheduling, register allocation, global optimization
`-tp {athlonxp | k8-32 | k8-64 | px | p5 | p6 | p7}`	Select a processor target
`-Munroll`	Unroll loops
`-fast`	Includes "`-O2 -tp <target> -Munroll -Mnoframe -Mlre`"
`-Mscalarsse`	Perform all scalar floating-point operations using SSE/SSE2
`-Mvect=sse`	Try to use SSE/SSE2 "packed" instructions to speed up vectorizable loops
`-fastsse`	Includes "`-fast -Mvect=sse -Mscalarsse -Mcache_align`"
`-Minline`	Inline functions and subroutines
`-Mconcur`	Try to auto-parallelize loops for SMP systems
`-mp`	Process OpenMP/SGI directives and pragmas
`-byteswapio`	Swap bytes from big-endian to little-endian or vice versa upon input/output of Fortran unformatted data files

NOTE: On AMD64 systems, `-Mscalarsse` is default; in the future it will become default on *all* AMD Athlon 64, AMD Opteron, Pentium 4 and Xeon processors and follow-ons

Important PGI Compiler Options (cont)

`-pc {32|64|80}, -Kieee`

Set precision of ops on FP stack, strict IEEE

`-Mcache_align`

Align unconstrained objects >= 16 bytes to a cache line boundary

`-Mneginfo={concur|loop}`

Display messages that indicate why loops are not parallelized or vectorized

`-Msecond_underscore, -g77libs`

g77 compatibility

`-Mipa=fast` Enable inter-procedural analysis and optimization; especially important for F90 and C

`-Minfo` Compile-time optimization/parallelization messages

`-Mprof` Enable function or line-level profiling

x87 vs SSE Floating-point Arithmetic

- By default, **all FP arithmetic** on AMD Athlon ™, PII and prior x86 targets is performed using **x87 FP stack** instructions, which by default are 80-bit IEEE; precision can be reduced to 64-bit IEEE or 32-bit IEEE globally using a mode bit (can set using the `-pc` flag)
- On AMD Athlon and PentiumIII, **all scalar FP arithmetic** is performed **using x87**, but **32-bit FP data vector loops** that can be performed **using SSE** instructions (IEEE 32-bit) if `-Mvect=sse` is specified
- On Pentium4/Xeon and AMD64, **all FP arithmetic** can be performed using either **x87** ***or*** **SSE/SSE2**; SSE/SSE2 provide "normal" 32-bit and 64-bit IEEE arithmetic
- Default on Pentium4/Xeon is x87; default on AMD64 is SSE/SSE2; `-Mscalarsse` forces SSE/SSE2; on either target, vector loops that operate on 32-bit or 64-bit FP data can be performed using SSE instructions (IEEE 32-bit and 64-bit) if `-Mvect=sse` is specified

There is a performance advantage to SSE, even for scalar arithmetic, and since it is pure 32/64-bit IEEE arithmetic, results will generally match most RISC/UNIX systems (Sun, SGI, IBM, etc)

Using the –Kieee and -pc Options w/x87

`-pc {32 | 64 | 80}`

Perform scalar FP operations on the x87 floating-point stack using the specified number of bits of precision; default is 80, with values being rounded when stored back to memory

`-Kieee`

Perform all FP ops in strict compliance to the IEEE 754 standard; disables copy propagation and reciprocal division, and makes function calls to perform all transcendental operations

Generally, –pc 64 or –pc 64 in combination with –Kieee Produces arithmetic results, even w/x87, that match most RISC/UNIX systems

And if all else fails …we just cheated

```
R8_var = R4_var * R4_var
```

No matter what you do w/-pc and –Kieee, the operation is done in 64-bit and assigned to 64-bit, or is done in 32-bit and assigned to 32-bit, but defining a function as follows will do the trick:

```
Interface operator (+)
    function add32(x,y)
    real*4 add32
    real*4 x, y
    end function
End interface
```

Even though it's illegal F90, PGF90 allows overloaded standard operators – saved one customer from having to comb through 450K lines of legacy fortran to fix such problems in order to get "right answers"

Streaming SIMD Enhancements (SSE)

- First available on Pentium III and AMD Athlon XP for 32-bit data (SSE), expanded to support 64-bit data (SSE2) in P4/Xeon; *not backward-compatible*
- Can be used to perform either scalar or "packed" operations
- **On Pentium4/Xeon/AMD64 32-bit** - *Eight 128-bit SSE registers* + new instructions to operate on them; essentially size 4 vector regs/instructions for 32-bit FP data or size 2 vector regs/instructions for 64-bit FP data
- **On AMD64/ 64-bit** – *Expanded to sixteen 128-bit SSE registers* + all previous instructions and capabilities
- "Vectors" must be 16-byte aligned for optimal performance
- PGI Compilers recognize vectorizable loops and generate code to perform operations 4-at-a-time (32-bit) or 2-at-a-time (64-bit) to increase performance on compute-intensive loops
- Force *all* FP arithmetic to be performed with SSE/SSE2 by using the `-fastsse` or `-Mscalarsse` option on targets that support SSE/SSE2; `-fastsse` => "`-fast -Mvect=sse -Mscalarsse -Mcache_align`"

Using the -Mvect Option

`-Mvect[=option[,option]]` where *option* is:

`altcode:n`	**Instructs the vectorizer to generate alternate scalar code for vectorized loops**
`noaltcode`	**Do not generate alternate scalar code**
`cachesize:n`	**Tile loops assuming cache size of `n` bytes, default `n` = 262144**
`prefetch`	**Generate software prefetch instructions**
`[no]sse`	**[Don't] Use SSE/SSE2 "packed" instructions wherever possible**
`[no]assoc`	**[Don't] Vectorize loops with reductions**

Vectorizable Loop in SPECFP2K FACEREC (Data is REAL*4)

```
350 !
351 !   Initialize vertex, similarity and coordinate arrays
352 !
353     Do Index = 1, NodeCount
354       IX = MOD (Index - 1, NodesX) + 1
355       IY = ((Index - 1) / NodesX) + 1
356       CoordX (IX, IY) = Position (1) + (IX - 1) * StepX
357       CoordY (IX, IY) = Position (2) + (IY - 1) * StepY
358       JetSim (Index)  = SUM (Graph (:, :, Index) * &
359     &                   GaborTrafo (:, :, CoordX(IX,IY), CoordY(IX,IY)))
360       VertexX (Index) = MOD (Params%Graph%RandomIndex (Index) - 1, NodesX) + 1
361       VertexY (Index) = ((Params%Graph%RandomIndex (Index) - 1) / NodesX) + 1
362     End Do
```

Inner loop at line 358 is vectorizable, can used packed SSE instructions

Use –Minfo to see Which Loops Vectorize

```
% pgf90 -tp k8-32 -fastsse -Mipa=fast -Minfo -Mkeepasm  -c graphRoutines.f90
…
localmove:
   334, Loop unrolled 1 times (completely unrolled)
   343, Loop unrolled 2 times (completely unrolled)
   358, Generating vector sse code for inner loop
   364, Generating vector sse code for inner loop
        Generating vector sse code for inner loop
   392, Generating vector sse code for inner loop
   423, Generating vector sse code for inner loop
%
```

And the Resulting Assembly Code …

Scalar SSE:

```
.LB6_668:
# lineno: 358
        movss   -12(%eax),%xmm1
        movss   -8(%eax),%xmm2
        movss   -4(%eax),%xmm3
        decl    %edx
        mulss   -12(%ecx),%xmm1
        addss   -572(%ebp),%xmm1
        mulss   -8(%ecx),%xmm2
        addss   %xmm2,%xmm1
        mulss   -4(%ecx),%xmm3
        addss   %xmm3,%xmm1
        movss   (%eax),%xmm2
        addl    $16,%eax
        mulss   (%ecx),%xmm2
        addl    $16,%ecx
        addss   %xmm2,%xmm1
        testl   %edx,%edx
        movss   %xmm1,-572(%ebp)
        jg      .LB6_668
```

Vector SSE:

```
.LB6_1105:
# lineno: 358
        movlps  (%esi,%ecx),%xmm2
        movlps  (%edx,%ecx),%xmm3
        movhps  8(%esi,%ecx),%xmm2
        movhps  8(%edx,%ecx),%xmm3
        mulps   %xmm2,%xmm3
        movlps  16(%esi,%ecx),%xmm2
        movhps  24(%esi,%ecx),%xmm2
        addps   %xmm3,%xmm0
        movlps  16(%edx,%ecx),%xmm3
        movhps  24(%edx,%ecx),%xmm3
        addl    $32,%ecx
        mulps   %xmm2,%xmm3
        addps   %xmm3,%xmm0
        subl    $8,%eax
        jg      .LB6_1105
```

OpenMP - Using the -mp Option

- Supports Full OpenMP 1.1 for F77/F90, and full OpenMP 1.0 for C/C++
- Also supports SGI C$DOACROSS in Fortran and SGI pragmas in C/C++
- Use *–Mnosgimp* to ignore SGI, and *–Mnoopenmp* to ignore OpenMP directives in programs that have both (e.g. MM5)

Using the -Mconcur Option

`-Mconcur[=`*`option`*`[,`*`option`*`]]` where *`option`* is:

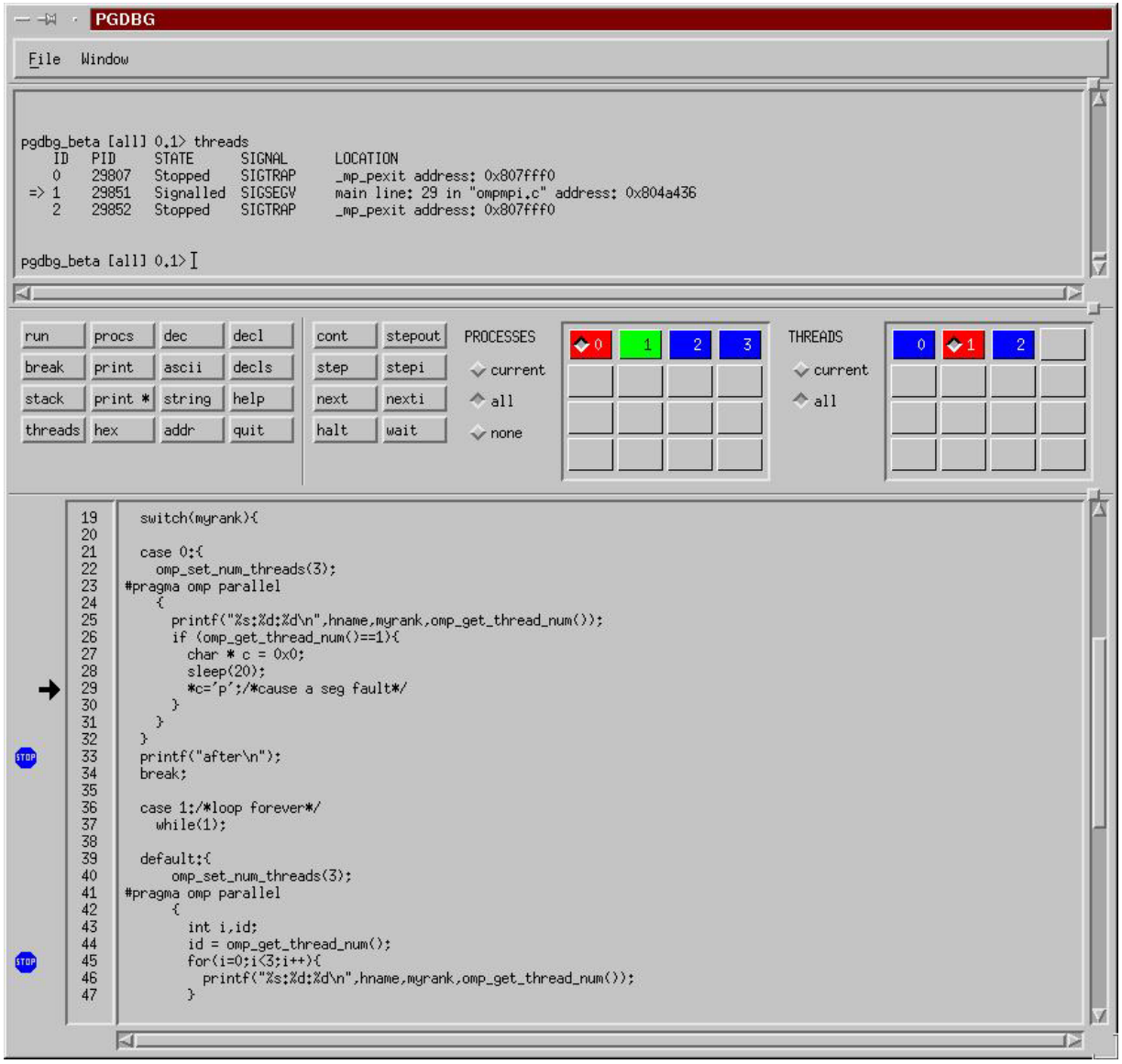
`altcode:n`	**Instructs the parallelizer to generate alternate scalar code for parallelized loops**
`noaltcode`	**Do not generate alternate scalar code**
`dist:block`	**Parallelize with block distribution (default)**
`dist:cyclic`	**Parallelize with cyclic distribution**
`cncall`	**Loops containing calls are candidates for parallelization**
`noassoc`	**Disables parallelization of loops with reductions**

Using the -Minline Option

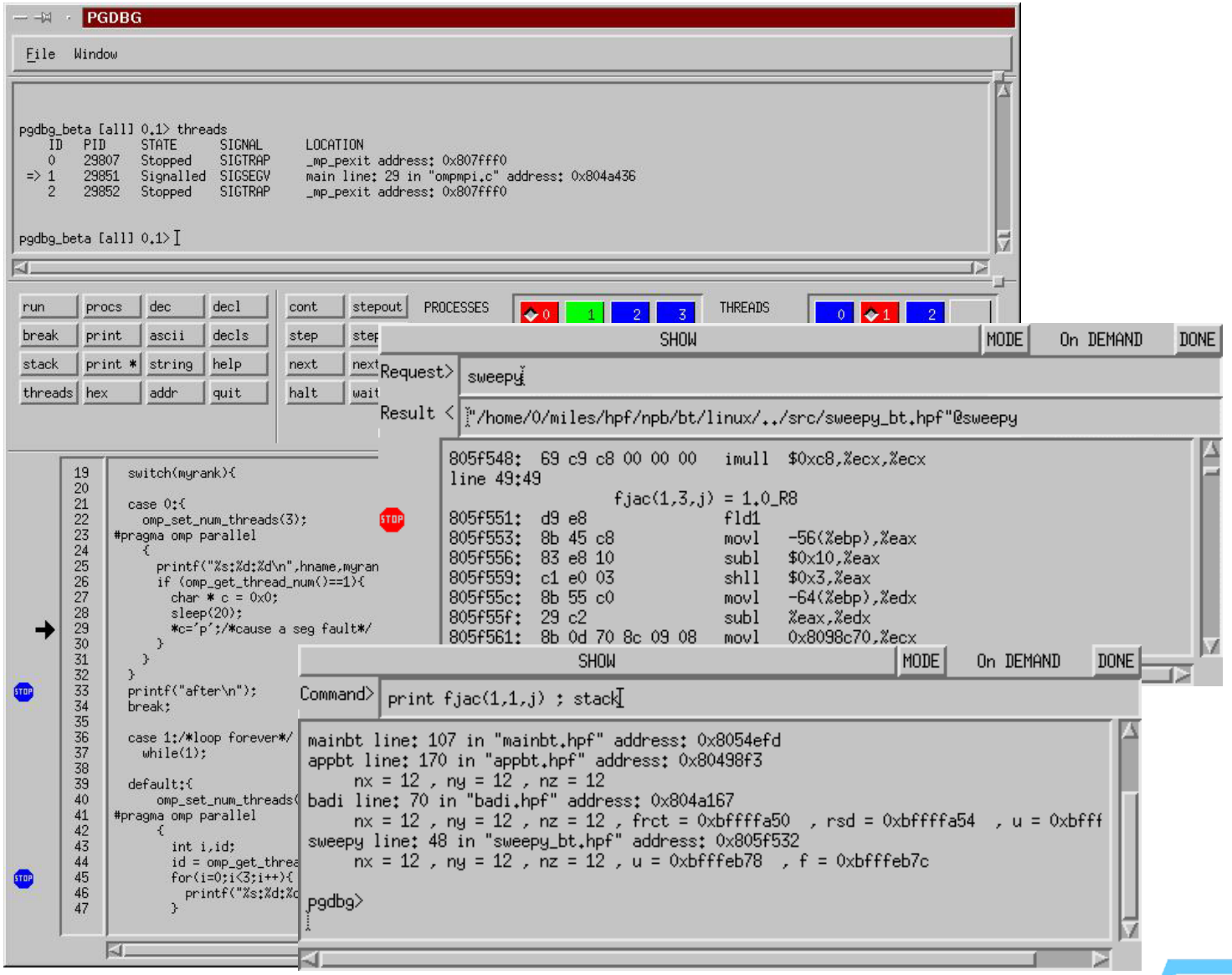
`-Minline[=`*`option`*`[,`*`option`*`]]` where *`option`* is:

`size:`**`n`**	**Instructs the inliner to inline functions or subroutines with `n` or fewer lines.**
`[name:]f1[,f2[…]]`	**Inline functions/subroutines with names `f1`, `f2`**
`levels:`**`n`**	**Instructs the inliner to inline functions up to `n` levels deep. The default value of `n` is 2.**
`[lib:]<`*`file.ext`*`>`	**Inline functions/subroutines from the inline library `file.ext`.**

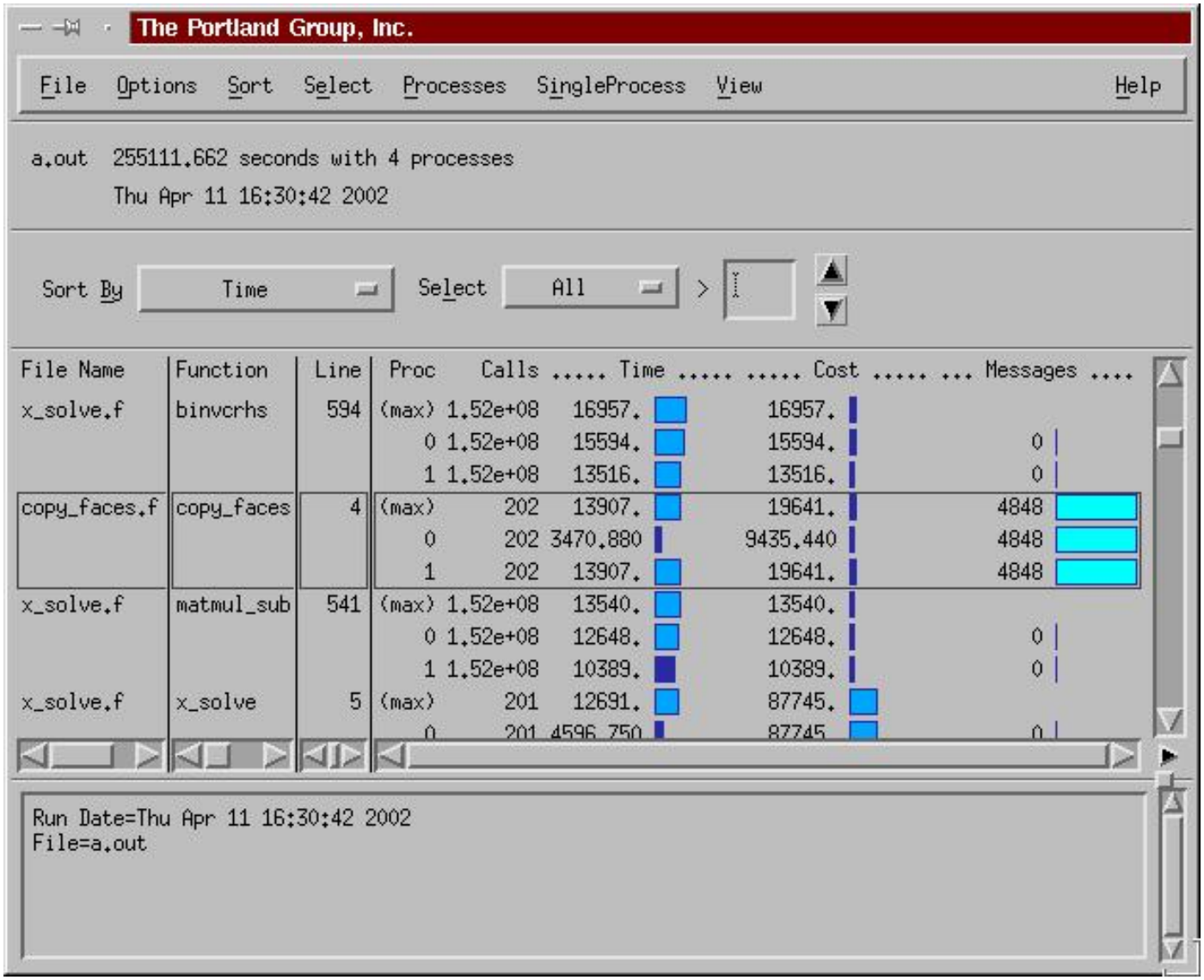
PGDBG Graphical OpenMP/MPI Debugger



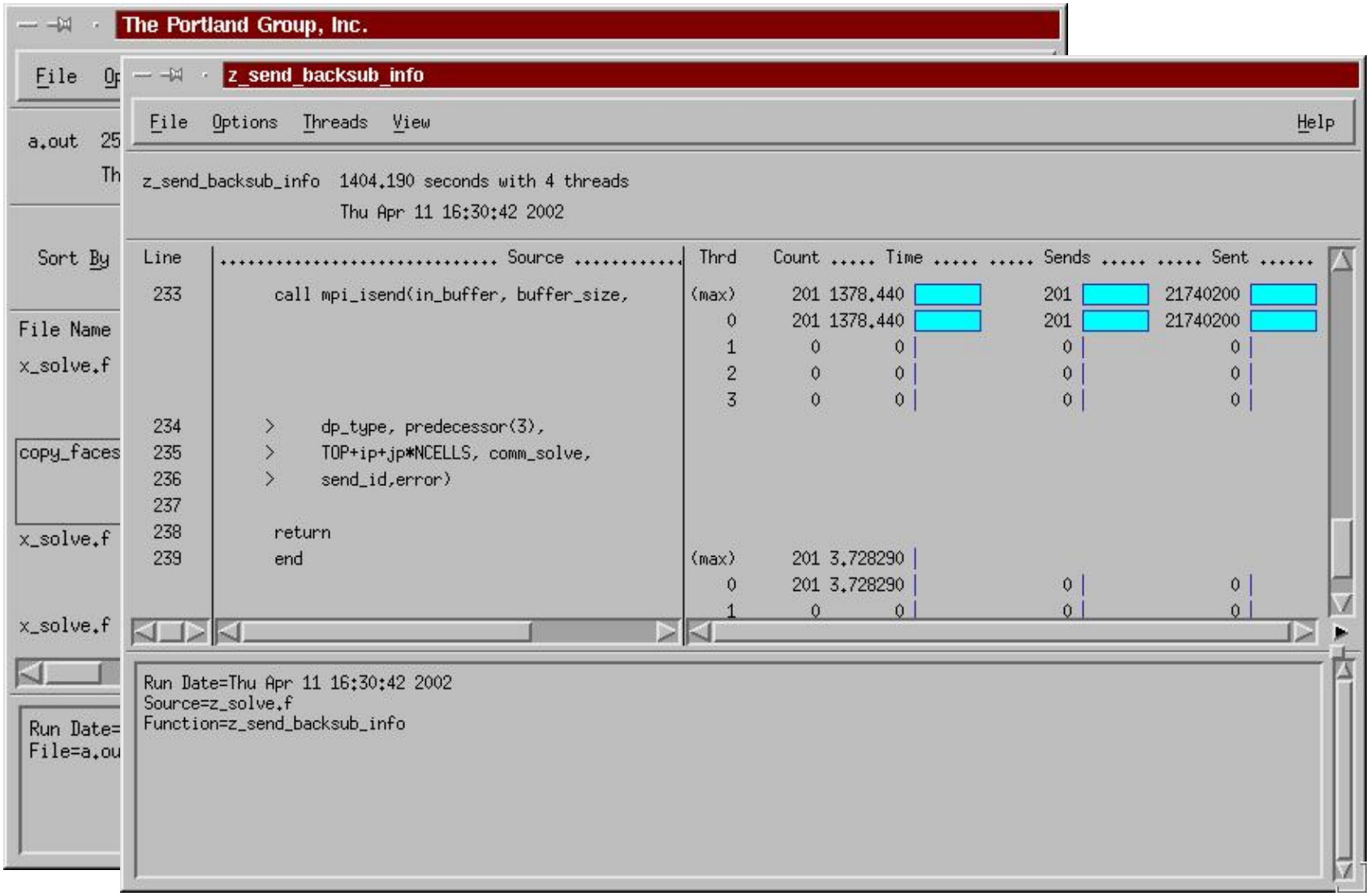
PGDBG Graphical OpenMP/MPI Debugger



PGPROF Graphical OpenMP/MPI Profiler



PGPROF Graphical OpenMP/MPI Profiler



Performance Tuning Summary

- **Migrate to SSE/SSE2:** Use `-fast -Mscalarsse` or `-fastsse` options
- **Vectorize:** If you are writing new code or tuning, try to formulate loops that vectorize efficiently – stride 1 accesses, compute intensity, do not unroll by hand (tiling OK if tiles are large); time your code to make sure it's faster
- **Interprocedural Analysis:** `-Mipa`, Try it! It can help on C codes and some F90 codes (pointer disambiguation, shape propagation), and almost never will hurt performance; will get easier in the future
- **Inlining:** Try inlining if your application contains large numbers of calls to small functions; this will reduce call overhead and increase opportunities for vectorization
- **Parallelize:** Try `-Mconcur` on an SMP system; time your code to make sure it's faster; Try OpenMP directives/pragmas
- **Use PGPROF and `-Minfo`:** Use PGPROF (compile/link `-Mprof=func`) to identify expensive functions; examine output of `-Minfo` to see which loops are vectorizing/parallelizing and which are not

PGI Compilers & Tools Futures

- Fully tuned AMD64 code generator (5.1 and 6.0)
- Completely new 32-bit x86 code generator; better integration of scalar and vector code generation (5.1)
- One-pass Inter-Procedural Analysis (IPA) for ease-of-use (6.0)
- Enhanced IPA optimizations for F90/F95 (5.1 and 6.0)
- Additional F95 features – user-defined elementals, pointer initialization, etc (6.0)
- Support for single data objects > 2GB (C/F77 5.1, C++/F90 6.0)
- C/C++ tuning (6.0)
- New cross-platform GUIs for PGDBG and PGPROF on Linux and Windows, including enhanced parallel capabilities (6.0)
- Improved scalability of PGDBG for Linux Clusters (TBD)
- Sample-based profiling on Linux (5.1)
- Integration with Windows Services For UNIX (SFU)?